ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA149

SOFTWARE AND GATEWARE DEVELOPMENT FOR SIRIUS BPM ELECTRONICS USING A SERVICE-ORIENTED ARCHITECTURE

L. M. Russo*, LNLS, Campinas, SP, Brazil

Abstract

The Brazilian Synchrotron Light Laboratory (LNLS) is in the final stages of developing an open-source BPM system for Sirius, a 4th-generation synchrotron light source under construction in Brazil. The system is based on the MicroTCA.4 standard comprising AMC FPGA boards carrying FMC digitizers and a CPU module. The software is built with the HALCS framework [1] and employs a service-oriented architecture (SOA) to export a flexible interface between the gateware modules and its clients, providing a set of loosely-coupled components favoring reusability, extensibility and maintainability. In this paper, the BPM system will be discussed in detail focusing on how specific functionalities of the system are integrated and developed in the framework to provide SOA services. In particular, two domains will be covered: (i) gateware modules, such as the ADC interface, acquisition engine and digital signal processing; (ii) software services counterparts, showing how these modules can interact with each other in a uniform way, easing integration with control systems.

INTRODUCTION

Many particle accelerators and high-energy physics experiments are based on a distributed industrial control system, such as: EPICS [2], Tango [3], DOOCS [4]. In order to fulfill the requirements of such demanding installations in terms of scalability, decoupling, reliability and evolution, their foundation lies on a two or three-tier architecture, effectively decoupling the so called *Front-End Controller* (*FEC*) layer, *Client Application* layer and, in the case of a three-tier architecture, *Middle-Layer Services* layer.

Hence, to accomplish these goals, two concepts are generally employed: (i) an *entity* that abstracts a given functionality, being either an actual equipment or an abstract computing service; (ii) a *contract*, specifying how to communicate with this entity in order to explore that exported functionality.

Examples of the *entity* concept are clear by looking at the existing control system toolkits, such as: *Input-Output Controller* (IOC) for EPICS and *Device Server* for Tango and DOOCS. As for the *contract* concept, EPICS defines the *Channel Access* protocol for EPICSv3 and the *PVAccess* protocol for EPICSv4, Tango establishes the contract on top of existing Message-based middleware technologies like ZeroMQ [5] and CORBA [6], and lastly, DOOCS uses the ONC/RPC protocol [7].

As a natural advancement of the underlying modern control system architecture, many applications have been developed by following what is called *Service-Oriented Architecture* (SOA) principles [8] [9], such as: loose-coupling between parts of the system, a standard contract between the parts, reusability, service abstraction, service autonomy, among others. Examples of this can be seen in [10–15].

By using the same SOA principles over which high-level applications have been successfully employed over the years, development and integration of systems using modular platforms like MicroTCA [16], can be leveraged by the use of the same approach. Specifically, but not limited to, applications that make use of FPGA chips can benefit a lot from this approach, so that even micro-modules can be abstracted as an *entity* and controlled by a common *contract*, favoring reusability and maintainability.

In the next sections, the tools that provide this abstraction, as well as the benefits and challenges, are going to be discussed.

SOA ARCHITECTURE FOR LOW-LEVEL SYSTEMS

Traditional SOA principles are based on ideas that favor independent design and evolution of each part of a system. In many cases, however, low-level systems controlling an embedded or external hardware device lack these features, even though they tend to follow a modular design philosophy.

The reason behind this might be due to the intrinsic tightly-coupled characteristic of these systems and the difficulties in achieving the end metrics, namely: latency of operations, number of operations per second and system scalability. Taking full advantage of the SOA principles in this case is harder, but not intangible, as the key idea is to dissociate the performance-driven part of the system, in which a more tightly-coupled design might be inevitable, from the actual architecture of the system, in which the actual properties and the interrelationships between the system's entities are described. The former generally needs to use a more complex protocol and abstractions, while the latter has the freedom to use SOA principles to increase modularity, maintainability and reuse.

Topology for Low-Level Systems

In a generic way, low-level systems, particularly FPGA-based with a CPU attached to it (either via PCIe, Ethernet or an internal bus), can be described in two parts. The first one is composed by the specific hardware to perform the end application, along with its peripherals, FPGA, ADCs, DACs and others. The second part is denoted by a controlling agent (typically a software composing a *FEC*) interacting with the hardware and providing an abstraction to other systems.

* lucas.russo@lnls.br

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA149

Figure 1 shows a generic hierarchical description of the hardware part and Fig. 2 depicts the corresponding SOA-based software for controlling that hardware.

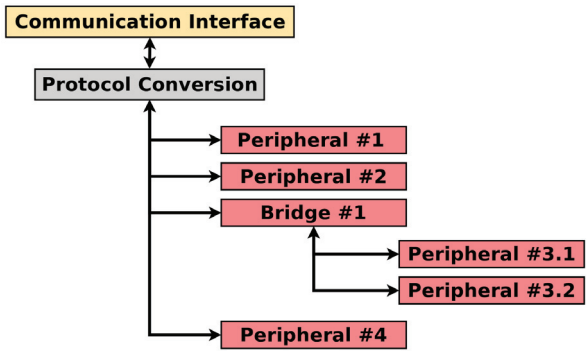

Figure 1: Generic Hardware Architecture.

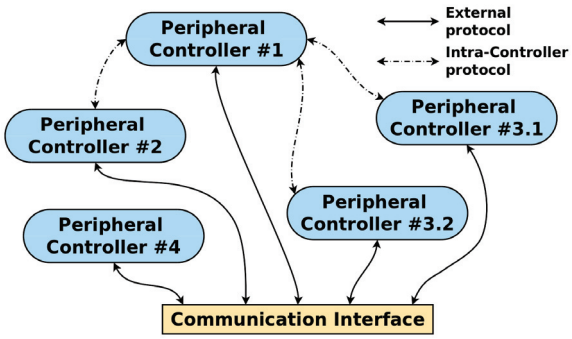

Figure 2: SOA-based Software Architecture.

The ability to structure the application by means of small abstract components and a stable protocol between them also favors the evolution of the software and can ease new design developments. As shown in Fig. 1, the hardware architecture is generally composed of a communication interface, by which the software can interact with it, a protocol conversion layer, to translate between the interface and an internal bus protocol, and a set of peripherals.

By having each small peripheral in hardware matching one service unit in software, as shown in Fig. 2, it can be thought as a composable abstraction layer, in which the individual services provide a specific functionality and can be aggregated to form more complex services.

Oftentimes, some inter-service dependencies occur in the design. To solve this, an *Intra-Controller protocol* can be used to enable communication between the services, so its actions can be coordinated to achieve the end functionalities.

Hardware Prerequisites

In order to achieve this, the hardware needs to be designed such that its internal components are functionally isolated, with minimal coupling between its parts and independently controlled. This is especially important to allow the software to abstract the hardware in terms of SOA services. Apart from the use of a standard bus protocol, such as Wishbone [17] and the employment of consistent interfaces for exchange data/controls, the project itself must support this degree of decoupling. Without it, maintaining a SOA architecture might become too difficult and hard to coordinate operations among different components.

A desirable feature is to make available the information about the internal components of a certain hardware, as a means to describe each one of its independent parts, capabilities and topology. With such a feature, the knowledge about which hardware device is being controlled and what capabilities it offers can be moved to upper layers of the *FEC*. The advantage of this is having a more uniform and generic abstraction layer and the possibility to use automatic detection of known components, so to have an out-of-the-box controllable device. This can be achieved by the use of a self-describing bus (*SDB*) [18], which describes all of the design components, in terms of a unique identification, name, address range, version and capabilities. The *SDB* acts as an addressable memory from the software point of view and has a fixed layout, making it easy to parse and to extract all of the components' properties.

Software Prerequisites

In the software domain, it needs to provide all of the infrastructure regarding: (i) module isolation; (ii) communication protocols to allow the interaction between the modules in a system and external entities (i.e., other software) running outside of the system; (iii) easiness of adding, removing or altering a module without affecting either the other modules or the existing software infrastructure, typically following the inversion of control design logic.

BPM PROJECT MODELED USING SOA PRINCIPLES

The Beam Position Monitor (BPM) project was modeled according to the SOA principles following the topology depicted in Fig. 1 and Fig. 2. The next sections will describe how the gateware and software parts of the BPM system were designed, outlining the perceived advantages over more traditional approaches and possible limitations.

Gateware

The BPM gateware was developed by modularizing the design in two sets of reusable cores and a project-specific repository, based on the open-source computer bus Wishbone. The first one is composed of acquisition modules, trigger interfaces, PCIe, DDR, FMC ADC interfaces, among others [19]. The second one is composed of DSP cores, adders, multipliers, DDS, filters, and specific modules composing the BPM algorithms [20]. The last repository contains the top-level design files and project-specific cores [21] and its structure can be seen in Fig. 3.

The design follows a hierarchical design written in VHDL / Verilog. At the top of the hierarchy, there is a PCIe interface and a Serial RS232 interface. They both pass through a protocol conversion to support Wishbone master transactions.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA149

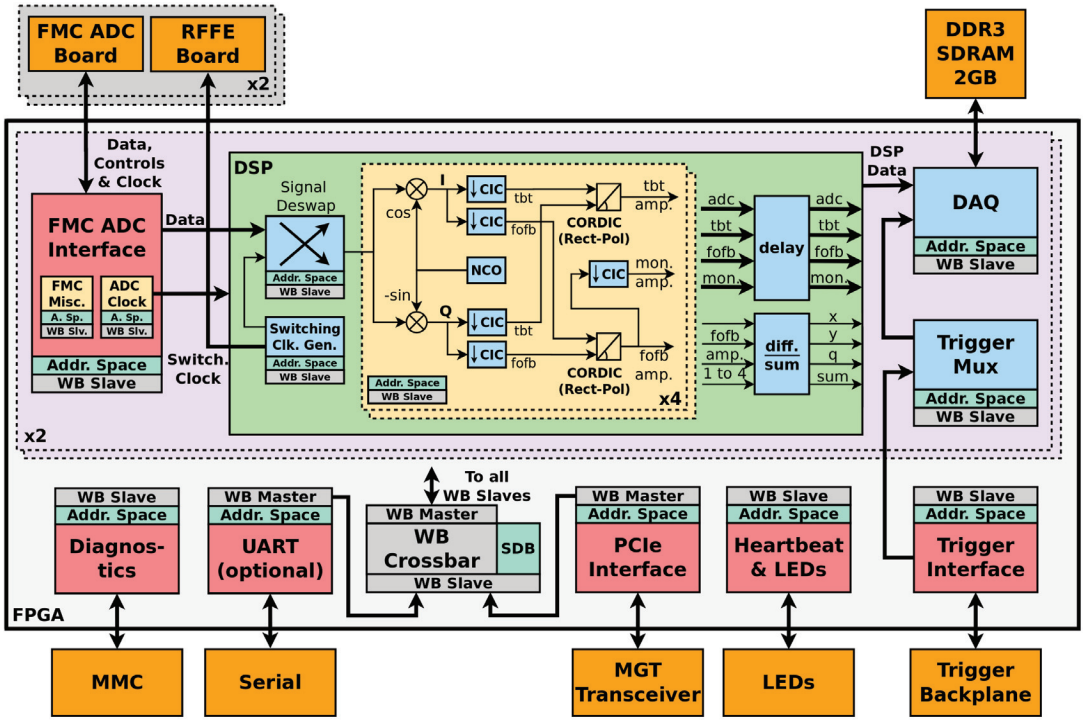

Figure 3: BPM Gateware Architecture.

As a core infrastructure module, there is a Wishbone Crossbar, implementing independent many-to-many connections between an arbitrary number of masters and slaves. Attached to it, the *SDB*, usually located at address 0x0 and implemented as a ROM, stores meta-information about the whole system, including all of the addressable components.

At the bottom of the hierarchy are the peripheral components, which could in turn include other *Wishbone* crossbars and peripherals, composing a nested architecture. The first component after the external *FMC ADC Board* is the FMC ADC interface itself. It is responsible for ADC data capture from 4 ADCs, data alignment, clock buffering and for controlling the *FMC ADC Board* peripherals, such as: Si57x programmable crystal oscillator, AD9510 clock distribution and PLL, ISLA216P ADC and 24AA64 EEPROM.

In the sequence, data from all 4 ADCs goes to a deswapping stage, in which signals from ADC channels 1 and 3 are swapped with signals from channels 2 and 4, respectively, at a frequency of ≈100 kHz. This implements the digital part of the switching scheme adopted by the BPM and has a crucial role in guaranteeing low-frequency (up to half the swapping frequency) noise suppression [22].

Afterwards, the signal is demodulated via an envelope detector to extract the amplitude of the 4 signals, which is then used to calculate the position of the beam by means of the delta-over-sum method.

The last stage of the BPM signal chain is composed of a multi-channel acquisition engine capable of acquiring all of the intermediate steps in the DSP chain, triggered either by an external signal, a data-driven trigger or a software trigger. It also supports an arbitrary number of pre- and post-trigger samples, up to the size of the external SDRAM memory. Additionally, a trigger multiplexer module selects a single trigger from its sources and outputs it to the respective acquisition channel, whereas the trigger interface module acts as a bridge between the external MLVDS pins available in the MicroTCA.4 backplane.

There is also some support modules included in the gateware, namely: Diagnostics and Heartbeat & LEDs. The first one is used to gather some information about the MicroTCA.4 sensors, such as slot number and I2C address. The second one is a simple 1 Hz heartbeat and basic LED control for visual feedback purposes.

Software

As for the software part a framework, called Hardware Abstraction Layer for Control Systems (*HALCS*) [23], was developed to take advantage of SOA principles. In a simplified description the framework provides:

1. Module design interface
2. Inter-module protocol
3. External RPC protocol
4. Communication interfaces (e.g., PCIe and TCP/IP)
5. Composable chip interfaces (e.g., I2C, SPI, GPIO)
6. Automatic module detection (*SDB* parsing)

In this sense, *HALCS* provides a set of services (or modules) controlled by a common RPC protocol, so that upper-level or client software (e.g., Control System or CLI programs) can perform its functions. So, in effect, the actual BPM software architecture is only known at runtime when the actual group of services, that are discovered by parsing the *SDB*, are spawned and their communication and

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA149

relationships are set up. Figure 4 shows the BPM software architecture for the previous BPM gateware of Fig. 3.

On software startup, the Hardware Abstraction Layer (HAL) (called *LLIO* in *HALCS*) is instantiated along with its hardware interface (PCIe, in the case of the BPM). The next step taken by *HALCS* is to try to find and parse the SDB structure within the device (if the SDB is not found, no module is spawned and the user needs to perform a manual instantiation). This is performed by the *Dispatch Engine* (called *DEVIO* in *HALCS*) and is responsible, among other things, to register/unregister modules (called *SMIO* in *HALCS*), send/receive control messages to/from *SMIOs* and to serialize external protocol messages coming from *SMIOs* for dispatching them to HAL.

The *External protocol*, implemented as an inter-thread communication protocol, used between *SMIOs* and *DEVIO*, is generally enough to completely abstract a functionality and the ability to control it in its entirety. This is used in most of the modules developed so far, as they are self-contained units that perform very specific functions, such as: Heartbeat & LEDs, Trigger Mux, Trigger Interface, DAQ, Diagnostics, FMC Misc and DSP. They don't rely on some intrinsic startup order or coordination from other modules to correctly operate. These services, therefore, ideally match the isolation and composability SOA principles.

However, some more complex modules, such as the FMC ADC Clock, FMC ADC and Deswap, need additional coordination. The reason for this is that FMC ADC Clock is responsible for configuring the FMC PLL and FMC clock oscillator, which drives all of logic in the ADC domain, particularly the ADCs and the deswap logic to the *RFFE* [22]. Hence, these modules cannot start or proceed beyond a certain point without extra confirmation. If done so, the modules might not start in a known state and can operate erratically.

For this reason, the FMC ADC Clock module, after completing the configuration of all of its components, communicates with the other two modules to inform them a stable clock is available and ready for use. On receiving this, the modules can proceed to its functions and perform the necessary actions, such as resetting some parts of the gateware, recalibrate some components or forcing a rewrite on some registers. If the clock or PLL needs to be reconfigured, following a request from a client for instance, the same actions can occur to satisfy this requirement.

This extra coordination is available by means of the *Intra-Controller protocol*, implemented in the same way as the *External protocol*, but through a different ZeroMQ socket, and can be used for generic message-passing among the modules. So far, only control messages were needed, but other use cases (e.g., periodic status messages) are certainly possible.

In order to communicate with its clients, the *SMIOs* are registered into a Broker, called *Malamute* [24], and its exported functionalities are made available. The Broker provides three important features: reliability, discoverability and an efficient low-level protocol between clients and services. The first one is accomplished by means of heartbeats, message confirmation, store-and-forward message mechanisms and automatic reconnection. Discoverability is important as the clients don't need to know the endpoints of each of the services, increasing client-service decoupling. Instead it can rely on a uniform string identifier and the broker will dispatch the message to the correct service. The last feature corresponds to a different protocol from the *External protocol*, and is suitable for efficient message-passing between nodes in a system, implementing many common patterns in distributed systems (i.e., mailbox, stream and publish-subscribe). Of particular interest is the mailbox pattern as it's particularly suitable for direct messaging between clients and services and is used for that matter.

Lastly, the clients communicate with the services using an RPC protocol on top of the low-level broker protocol. Two of the most important clients worth mentioning: (i) a set of command-line programs to interact with respective services, usually used to debug the system or to perform simple interactions with the services; (ii) the BPM EPICS driver [25], implemented by mapping the RPC functions to a set of associated EPICS database records.

BENEFITS OF USING SOA PRINCIPLES

The modularization of the design with these principles, and in particular by using a framework with characteristics such as those of *HALCS*, brings many advantages over more traditional approaches.

Usually, in order to support a new FPGA design or hardware, the software must be adapted to the new modules, addresses, protocols and communication interfaces. This incurs work on the low-level part of the software, which takes care of abstracting some FPGA module details. Moreover, the software clients or applications generally needs rework as they are either embedded in the same code as the low-level software or has a non-generic interface unsuitable for reuse. Even designs that take a more modular approach suffers from similar drawbacks, rendering the low-level software difficult to reuse, as it usually controls or abstracts more than just one component.

The abstraction of FPGA as a set of self-contained components and its corresponding software module services (or *SMIOs*) works in a way to provide clients with a suitable abstraction of the component, while not establishing policies. In this way, the services in *HALCS* could be thought as a user-space driver with an RPC interface and the actual application pushed over to the client layer, above the *Malamute Broker*.

Hence, the establishment of internal framework protocols, such as the *External protocol* and *Intra-Controller protocol*, provides clean and efficient interfaces to the modules, by using the fast ZeroMQ inter-thread message-passing, and encourages the module developer to design it in a reusable fashion. Coordination between modules can be accomplished by using the *Intra-Controller protocol*, adhering to the SOA principles.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA149

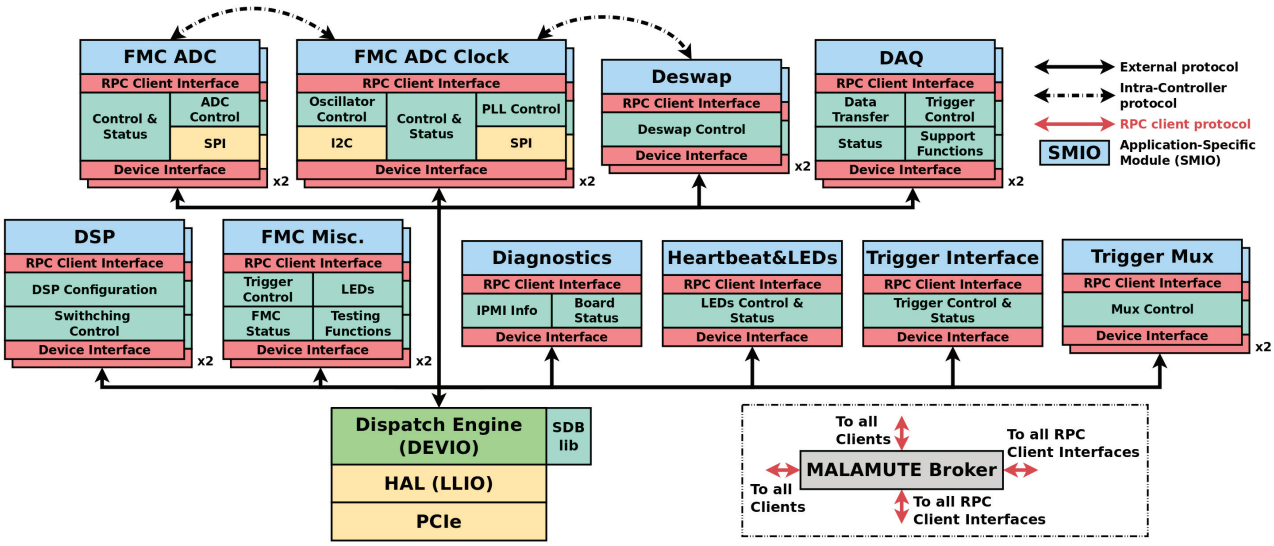

Figure 4: BPM Software Architecture.

Together with these benefits, the usage of *SDB* is of much convenience when developing new gateware designs, as the already supported software services (e.g., DAQ, Diagnostics, Trigger, LEDs) will be automatically spawned if a gateware component matches a known *SDB* ID and, in turn, its services will be exported to the clients with the same interface.

The current development of a new Sirius Timing Receiver gateware [26] for MicroTCA.4 platform is one example of these features. By using the same DAQ, Trigger and Diagnostics gateware components, it was possible to immediately use these functionalities and start testing the system without software modifications.

FUTURE WORK

In order to increase modularity of the *HALCS* framework and to allow a faster and easier integration of new *SMIOs*, a way to dynamically load new *SMIOs* without having to recompile the software is needed. Currently, this is necessary as the framework needs to know, at compile time, the *SDB* ID of all known components to build the known components' table. This is, of course, not ideal and requires new developers to modify the internal build system to include the new service. To achieve this, a simple approach might be to use the `dlopen()/dlsym()/dlclose()` family of functions. With them, it is possible to dynamically load new services by building them as a shared library that's independent of the *HALCS* sources.

As for the protocol communication interface both *External protocol* and *Intra-Controller protocol* used within *HALCS* are based on inter-thread communication. This is not completely adhering to the SOA principles as it partly violates the loose-coupling principle, in the sense that *entities* related to the same physical hardware device are not physically independent. To tackle this problem, an option would be to switch both protocols to the same *RPC protocol* already in use for communication with external clients, but this would incur a longer protocol processing latency and decreased bandwidth.

CONCLUSION

The principles and benefits of using a SOA approach to a low-level software/hardware domain were presented, with a special focus on the main project using it called Sirius BPM project, as well as a framework for achieving and endorsing these same principles, named *HALCS*. New projects, such as the upcoming MicroTCA.4 timing receiver gateware, can use the developed infrastructure and gathered knowledge to save time and design effort.

ACKNOWLEDGEMENT

The author of this paper would like thank Juan David Gonzalez Cobas and Federico Vaga, from CERN's BE-CO-HT section, and Xavi Serra and Manolo Broseta, from ALBA Synchrotron, for insightful discussions, Andrzej Wojeński, from the Warsaw University of Technology, for the initial gateware diagnostics component code, Henrique Almeida, from LNLS' Beamline Software Group, for reviewing important concepts, and the ZeroMQ community for the continuous delivery of quality code and support.

REFERENCES

[1] L. M. Russo, J. V. F. Filho, "Gateware and Software Frameworks for Sirius BPM Electronics", presented at PCaPAC'16, Campinas, Brazil, Oct. 2016, paper THDAPLCO03, unpublished.

[2] Experimental Physics and Industrial Control System (EPICS), `http://www.aps.anl.gov/epics`

[3] Tango Controls Toolkit, `http://www.tango-controls.org`

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA149

[4] Distributed Object Oriented Control System (DOOCS) Toolkit, http://tesla.desy.de/doocs/doocs.html

[5] ZeroMQ Messaging Library Project, http://zeromq.org

[6] Common Object Request Broker Architecture (CORBA), http://www.corba.org

[7] ONC/RPC Protocol, http://docs.oracle.com/cd/E19683-01/816-1435/rpcpguide-64802/index.html

[8] Service-Oriented Architecture (SOA), http://www.opengroup.org/standards/soa

[9] T. Erl, *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*, Upper Saddle River, NJ, USA: Prentice Hall PTR, 2007.

[10] P. Chu *et al.*, "Accelerator Lattice and Model Services", in *Proc. ICALEPCS'13*, San Francisco, USA, Oct. 2013, paper MOPPC152, pp. 464–466.

[11] N. Kulabukhova, A. Ivanov, V. Korkhov, A. Lazarev, "Software for Virtual Accelerator Designing", in *Proc. ICALEPCS'11*, Grenoble, France, Oct. 2011, paper WEPKS016, pp. 816–818.

[12] J. Chrin, M. Aiba, A. Rawat, Z. Wang, "Accelerator Modelling and Message Logging with ZeroMQ", in *Proc. ICALEPCS'15*, Melbourne, Australia, Oct 2015, paper WEB3O04, pp. 610–614.

[13] W. Pessemier, G. Raskin, H. Van Winckel, G. Deconinck, P. Saey, "A Practical Approach to Ontology-Enabled Control Systems for Astronomical Instrumentation", in *Proc. ICALEPCS'13*, San Francisco, USA, Oct. 2013, paper TUCOCB03, pp. 952–955.

[14] N. Wang, S. Shasharina, J. Matykiewicz, R. Pundaleeka, "Experiences in Messaging Middleware for High-Level Control Applications", in *Proc. ICALEPCS'11*, Grenoble, France, Oct. 2011, paper WEPKN005, pp. 720–723.

[15] L. Dalesio *et al.*, "Distributed Information Services for Control Systems", in *Proc. ICALEPCS'13*, San Francisco, USA, Oct. 2013, paper WECOBA02, pp. 1000–1003.

[16] MicroTCA PICMG Standard, https://www.picmg.org/openstandards/microtca

[17] Wishbone Computer Bus, http://opencores.org/howto/wishbone

[18] Self-Describing Bus Project, www.ohwr.org/projects/fpga-config-space

[19] Infra Cores Project, https://github.com/lnls-dig/infra-cores

[20] DSP Cores Project, https://github.com/lnls-dig/dsp-cores

[21] Beam Position Monitor Gateware Project, https://github.com/lnls-dig/bpm-gw

[22] R. A. Baron, F. H. Cardoso, J. L. B. Neto, S. R. Marques, J.-C. Denard, "Development of the RF Front End Electronics for the Sirius BPM System", in *Proc. IBIC'13*, Oxford, UK, Sep. 2013, paper WEPC07, pp. 670–673.

[23] Hardware Abstraction Layer for Control Systems Project, https://github.com/lnls-dig/halcs

[24] Malamute ZeroMQ Messaging Broker Project, https://github.com/zeromq/malamute

[25] BPM EPICS IOC, https://github.com/lnls-dig/bpm-epics-ioc

[26] Timing Receiver Project, https://github.com/lnls-dig/tim-receiver-gw